The eXpress Data Path

In this article I review the eXpress Data Path, the new kernel component for fast packet processing

Posted by Diego Pino García on January 10, 2019

In the previous article (https://blogs.igalia.com/dpino/2019/01/07/introduction-to-xdp-and-ebpf/) I briefly introduced XDP (*eXpress Data Path*) and eBPF, the multipurpose in-kernel virtual machine. On the XDP side, I focused only on the motivations behind this new technology, the reasons why rearchitecting the Linux kernel networking layer to enable faster packet processing. However, I didn't get much into the details on how XDP works. In this new blog post I try to go deeper into XDP.

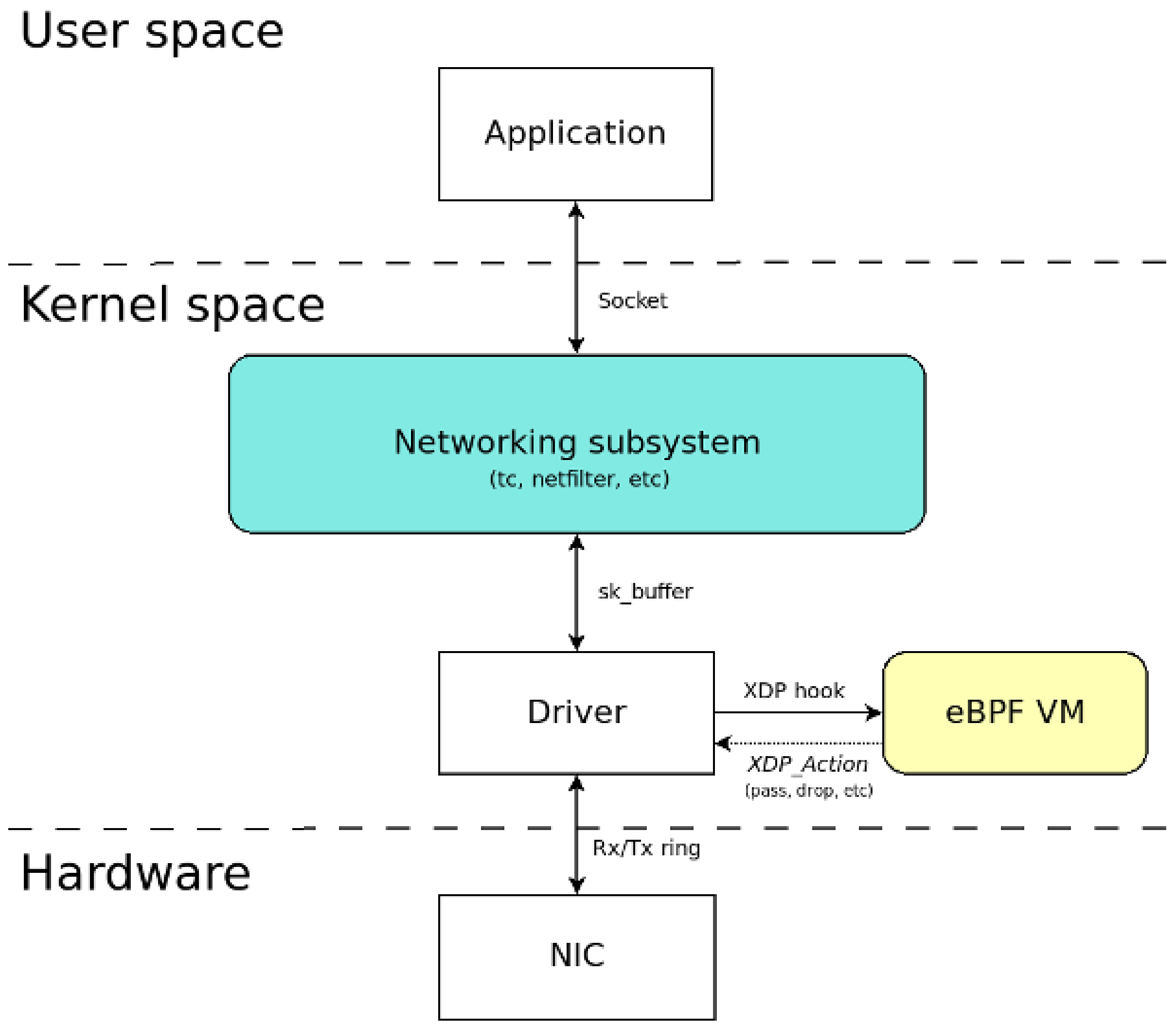
XDP: A fast path for packet processing

The design of XDP has its roots in a DDoS attack mitigation solution presented by Cloudflare at Netdev 1.1. Cloudflare leverages heavily on `iptables`, which according to their own metrics is able to handle 1 Mpps on a decent server (Source: Why we use the Linux kernel's TCP stack (https://blog.cloudflare.com/why-we-use-the-linux-kernels-tcp-stack/)). In the event of a DDoS attack, the amount of spoofed traffic can be up to 3 Mpps. Under those circumstances, a Linux box starts to be overflooded by IRQ interruptions until it becomes unusable.

Because Cloudflare wanted to keep the convenience of using `iptables` (and the rest of the kernel's network stack), they couldn't go with a solution that takes full control of the hardware, such as DPDK. Their solution consisted of implementing what they called a "*partial kernel bypass*". Some queues of the NIC are still attached to the kernel while others are attached to an *user-space* program that decides whether a packet should be dropped or not. By dropping packets at the lowest point of the stack, the amount of traffic that reaches the kernel's networking subsystem gets significantly reduced.

Cloudflare's solution used the Netmap toolkit to implement its partial kernel bypass (Source: Single Rx queue kernel bypass with Netmap (https://blog.cloudflare.com/single-rx-queue-kernel-bypass-with-netmap/)). However this idea could be generalized by adding a checkpoint in the Linux kernel network stack, preferably as soon as a packet is received in the NIC. This checkpoint should pass a packet to an *user-space* program that will decide what to do with it: drop it or let it continue through the normal path.

Luckily, Linux already features a mechanism that allows *user-space* code execution within the kernel: the eBPF VM. So the solution seemed obvious.



Linux network stack with XDP

Packet operations

Every network function, no matter how complex it is, consists of a series of basic operations:

- **Firewall**: read incoming packets, compare them to a table of rules and execute an action: *forward* or *drop*.
- **NAT**: read incoming packets, modify headers and forward packet.
- **Tunelling**: read incoming packets, create a new packet, embed packet into new one and forward it.

XDP passes packets to our eBPF program which decides what to do with them. We can read them or modify them if we need it. We can also access to helper functions to parse packets, compute checksums, and other functionalities, at no cost (avoiding system call cost penalties). And thanks to eBPF Maps we have access to complex data structures for persistent data storage, like tables. We are also able to decide what to do with a packet. Are we going to drop it? Forward it? To control a packet's processing logic, XDP provides a set of predefined actions:

- **XDP_PASS**: pass the packet to the normal network stack.
- **XDP_DROP**: very fast drop.
- **XDP_TX**: forward or TX-bounce back-out same interface.
- **XDP_REDIRECT**: redirects the packet to another NIC or CPU.
- **XDP_ABORTED**: indicates eBPF program error.

XDP_PASS, *XDP_TX* and *XDP_REDIRECT* are specific cases of a forwarding action, whereas *XDP_ABORTED* is actually treated as a packet drop.

Let's take a look at one example that uses most of these elements to program a simple network function.

Example: An IPv6 packet filter

The canonical example when introducing XDP is a DDoS filter. What such network function does is to drop packets if they're coming from a suspicious origin. In my case, I'm going with something even simpler: a function that filters out all traffic except IPv6.

The advantage of this simpler function is that we don't need to manage a list of suspicious addresses. Our program will simply examine the *ethertype* value of a packet and let it continue through the network stack or drop it depending on whether is an IPv6 packet or not.

```
SEC("prog")
int xdp_ipv6_filter_program(struct xdp_md *ctx)
{
    void *data_end = (void *)(long)ctx->data_end;
    void *data     = (void *)(long)ctx->data;
    struct ethhdr *eth = data;
    u16 eth_type = 0;

    if (!(parse_eth(eth, data_end, eth_type))) {
        bpf_debug("Debug: Cannot parse L2\n");
        return XDP_PASS;
    }

    bpf_debug("Debug: eth_type:0x%x\n", ntohs(eth_type));
    if (eth_type == ntohs(0x86dd)) {
        return XDP_PASS;
    } else {
        return XDP_DROP;
    }
}
```

The function `xdp_ipv6_filter_program` is our main program. We define a new section in the binary called *prog*. This serves as a hook between our program and XDP. Whenever XDP receives a packet, our code will be executed.

`ctx` represents a context, a `struct` which contains all the data necessary to access a packet. Our program calls `parse_eth` to fetch the `ethertype`. Then checks whether its value is *0x86dd* (IPv6 ethertype), in that case the packet passes. Otherwise the packet is dropped. In addition, all the `ethertype` values are printed for debugging purposes.

`bpf_debug` is in fact a macro defined as:

```
#define bpf_debug(fmt, ...)                              \
    ({                                                   \
        char ____fmt[] = fmt;                            \
        bpf_trace_printk(____fmt, sizeof(____fmt),       \
            ##__VA_ARGS__);                              \
    })
```

It uses the function `bpf_trace_printk` under the hood, a function which prints out messages in */sys/kernel/debug/tracing/trace_pipe*.

The function `parse_eth` takes a packet's beginning and end and parses its content.

```
static __always_inline
bool parse_eth(struct ethhdr *eth, void *data_end, u16 *eth_type)
{
    u64 offset;

    offset = sizeof(*eth);
    if ((void *)eth + offset > data_end)
        return false;
    *eth_type = eth->h_proto;
    return true;
}
```

Running external code in the kernel involves certain risks. For instance, an infinite loop may freeze the kernel or a program may access an unrestricted area of memory. To avoid these potential hazards a verifier is run when the eBPF code is loaded. The verifier walks all possible code paths, checking our program doesn't access out-of-range memory and there are not out of bound jumps. The verifier also ensures the program terminates in finite time.

The snippets above conform our eBPF program. Now we just need to compile it (Full source code is available at: xdp_ipv6_filter (https://github.com/dpino/xdp_ipv6_filter)).

```
$ make
```

Which generates `xdp_ipv6_filter.o`, the eBPF object file.

Now we're going to load this object file into a network interface. There are two ways to do that:

- Write an *user-space* program that loads the object file and attaches it to a network interface.

- Use `iproute2` to load the object file to an interface.

For this example, I’m going to use the latter method.

Currently there’s a limited amount of network interfaces that support XDP (*ixgbe*, *i40e*, *mlx5*, *veth*, *tap*, *tun*, *virtio_net* and others), although the list is growing. Some of this network interfaces support XDP at driver level. That means, the XDP hook is implemented at the lowest point in the networking layer, just when the NIC receives a packet in the Rx ring. In other cases, the XDP hook is implemented at a higher point in the network stack. The former method offers better performance results, although the latter makes XDP available for any network interface.

Luckily, `veth` interfaces are supported by XDP. I’m going to create a `veth` pair and attach the eBPF program to one of its ends. Remember that a `veth` always comes in pairs. It’s like a virtual patch cable connecting two interfaces. Whatever is transmited in one of the ends arrives to the other end and viceversa.

```
$ sudo ip link add dev veth0 type veth peer name veth1
$ sudo ip link set up dev veth0
$ sudo ip link set up dev veth1
```

Now I attach the eBPF program to `veth1`:

```
$ sudo ip link set dev veth1 xdp object xdp_ipv6_filter.o
```

You may have noticed I called the section for the eBPF program "*prog*". That's the name of the section `iproute2` expects to find and naming the section with a different name will result into an error.

If the program was successfully loaded I should see an `xdp` flag in the `veth1` interface:

```
$ sudo ip link sh veth1
8: veth1@veth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 xdp qdisc noqueue state UP mode DEFAULT group default qlen
1000
    link/ether 32:05:fc:9a:d8:75 brd ff:ff:ff:ff:ff:ff
    prog/xdp id 32 tag bdb81fb6a5cf3154 jited
```

To verify my program works as expected, I'm going to push a mix of IPv4 and IPv6 packets to `veth0` (`ipv4-and-ipv6-data.pcap`). My sample has a total of 20 packets (10 IPv4 and 10 IPv6). Before doing that though, I'm going to launch a `tcpdump` program on `veth1` which is ready to capture only 10 IPv6 packets.

```
$ sudo tcpdump "ip6" -i veth1 -w captured.pcap -c 10
tcpdump: listening on veth1, link-type EN10MB (Ethernet), capture size 262144 bytes
```

Send packets to `veth0`:

```
$ sudo tcpreplay -i veth0 ipv4-and-ipv6-data.pcap
```

The filtered packets arrived at the other end. The `tcpdump` program terminates since all the expected packets were received.

```
10 packets captured
10 packets received by filter
0 packets dropped by kernel
```

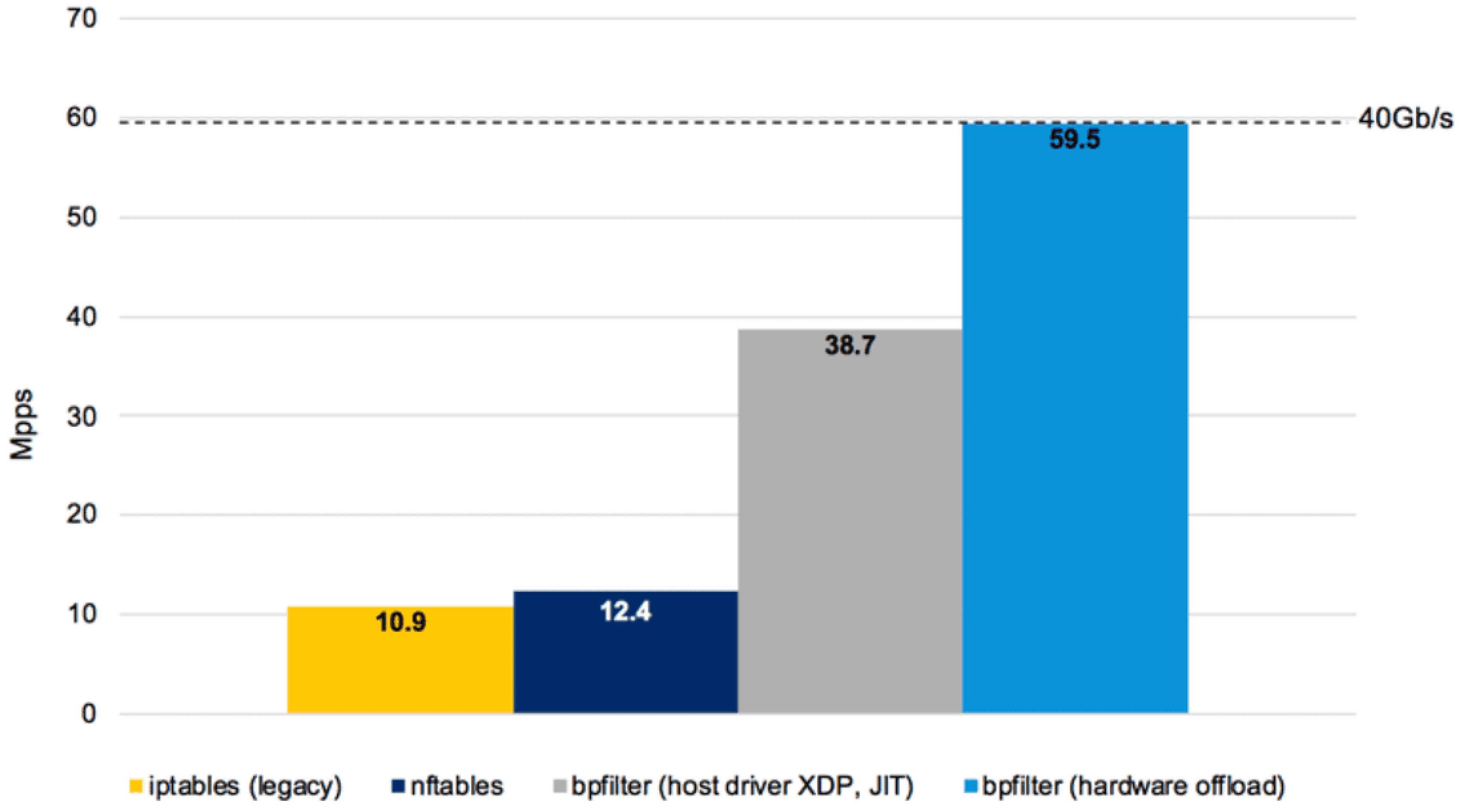
We can also print out `/sys/kernel/debug/tracing/trace_pipe` to check the ethertype values listed:

```
$ sudo cat /sys/kernel/debug/tracing/trace_pipe
tcpreplay-4496  [003] ..s1 15472.046835: 0: Debug: eth_type:0x86dd
tcpreplay-4496  [003] ..s1 15472.046847: 0: Debug: eth_type:0x86dd
tcpreplay-4496  [003] ..s1 15472.046855: 0: Debug: eth_type:0x86dd
tcpreplay-4496  [003] ..s1 15472.046862: 0: Debug: eth_type:0x86dd
tcpreplay-4496  [003] ..s1 15472.046869: 0: Debug: eth_type:0x86dd
tcpreplay-4496  [003] ..s1 15472.046878: 0: Debug: eth_type:0x800
tcpreplay-4496  [003] ..s1 15472.046885: 0: Debug: eth_type:0x800
tcpreplay-4496  [003] ..s1 15472.046892: 0: Debug: eth_type:0x800
tcpreplay-4496  [003] ..s1 15472.046903: 0: Debug: eth_type:0x800
tcpreplay-4496  [003] ..s1 15472.046911: 0: Debug: eth_type:0x800
...
```

XDP: The future of in-kernel packet processing?

XDP started as a fast path for certain use cases, especially the ones which could result into an early packet drop (like a DDoS attack prevention solution). However, since a network function is nothing else but a combination of basic primitives (reads, writes, forwarding, dropping…), all of them available via XDP/eBPF, it could possible to use XDP for more than packet dropping. It could be used, in fact, to implement any network function.

So what started as a fast path gradually is becoming the normal path. We're seeing now how tools such as `iptables` are getting rewritten in XDP/eBPF, keeping their user-level interfaces intact. The enormous performance gains of this new approach makes the effort worth it. And since the hunger for more performance gains never ends, it seems reasonable to think that any other tool that can be possibly written in XDP/eBPF will follow a similar fate.



iptables vs nftables vs bpfilter

Source: Why is the kernel community replacing iptables with BPF? (https://cilium.io/blog/2018/04/17/why-is-the-kernel-community-replacing-iptables/)

Summary

In this article I took a closer look at XDP. I explained the motivations that lead to its design. Through a simple example, I showed how XDP and eBPF work together to perform fast packet processing inside the kernel. XDP provides check points within the kernel's network stack. An eBPF program can hook to XDP events to perform an operation on a packet (modify its headers, drop it, forward it, etc).

XDP offers high-performance packet processing while maintaining interoperatibility with the rest of networking subsystem, an advantage over full kernel bypass solutions. I didn't get much into the internals of XDP and how it interacts with other parts of the networking subsystem though. I encourage checking the first two links in the recommended readings section for further understanding on XDP internals.

In the next article, the last in the series, I will cover the new AF_XDP socket address family and the implementation of a Snabb bridge for this new interface.

Recommended readings:

- The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel (https://raw.githubusercontent.com/tohojo/xdp-paper/master/xdp-the-express-data-path.pdf) by **Høiland-Jørgensen**, **Dangaard Brouer** and others. Paper published at CoNext '18 (December 4-7th, 2018). Possibly the best document published so far about XDP.

- XDP - challenges and future work (http://vger.kernel.org/lpc_net2018_talks/presentation-lpc2018-xdp-future.pdf) by **Jesper Dangaard Brouer** & **Toke Høiland-Jørgensen**. Talk at LPC (Linux Plumbers Conference) Networking Track. Status review of XDP and future plans. Complementary to the paper above.
- How to filter packets super fast: XDP & eBPF! (https://jvns.ca/blog/2017/04/07/xdp-bpf-tutorial/) by **Julia Evans**. Another great tutorial from Julia. These are the notes of an eBPF/XDP tutorial carried by Jesper D. Bouer and Magnuss Karlsson at NetDev 2.2 2018 (Montreal).
- Achieving high-performance, low-latency networking with XDP: Part I (https://developers.redhat.com/blog/2018/12/06/achieving-high-performance-low-latency-networking-with-xdp-part-1/) by **Paolo Abeni**. It explains how to load an eBPF program directly with iproute2. If you liked it, check out the follow up (https://developers.redhat.com/blog/2018/12/17/using-xdp-maps-rhel8/).

Update (10/06/2020):

Unfortunately, I never wrote the following article in this series.

Work on providing an XDP driver for Snabb was started at Snabb#1417 (https://github.com/snabbco/snabb/pull/1417) and a first implementation was available by the end of February 2019. The implementation relied heavily on some of the XDP examples available as part of the kernel sources. Performance wasn't as great as I expected and soon I found myself focused on other things.

Luckily, a few months after, Max Rottenkolber (https://twitter.com/eugeneia_) took over this task. He implemented XDP support for Snabb from scratch, developing in the meantime other interesting tools and contributing significantly to the state of XDP in the Linux kernel. Max summarized all this work in a great blog post, which can be considered the successor in spirit to this series. Check it out here: How to XDP (with Snabb) (https://mr.gy/blog/snabb-xdp.html).

I also recommend checking out Max's talk at Fosdem 2020 (https://fosdem.org/2020/schedule/event/vita_high_speed_traffic_encryption_on_x86_64/) about his work on Vita, a high-performance site-to-site VPN gateway based on Snabb. XDP is also covered in his talk, although briefly.

networking (/dpino/tag/networking) igalia (/dpino/tag/igalia)

← **PREVIOUS POST (/DPINO/2019/01/07/A-BRIEF-INTRODUCTION-TO-XDP-AND-EBPF/)**

NEXT POST → (/DPINO/2020/06/11/RENDERIZATION-OF-CONIC-GRADIENTS/)

● (/dpino/feed.xml) ● (https://twitter.com/diepg) ● (https://github.com/dpino)